Inductive Bases and their Application to Searches for Minimal Unary NFAs

Geoffrey Smith
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
smithg@cis.fiu.edu

ABSTRACT

Classic results in the theory of regular languages show that the problem of converting an NFA (nondeterministic finite automaton) into a minimal equivalent NFA is NP-hard, even for NFAs over a unary alphabet. This paper describes work on fast search techniques for finding minimal NFAs. The foundation of our approach is a characterization theorem for NFAs: we prove that a language is recognized by an n-state NFA iff it has what we call an *inductive basis* of size n. Using this characterization, we develop a fast incremental search for minimal NFAs for unary languages. We study the performance of our search algorithm experimentally, showing that, as compared with exhaustive search, it cuts the search space dramatically.

1. INTRODUCTION

In the theory of regular languages, the classic Myhill-Nerode theorem shows that every regular language has a unique minimal DFA (deterministic finite automaton) recognizing it. Moreover, this minimal DFA can be calculated efficiently: an n-state DFA can be converted into the minimal equivalent DFA in $O(n \log n)$ time [1]. In contrast, NFAs (nondeterministic finite automata) do not enjoy these nice properties. First, a regular language can have more than one minimal NFA recognizing it. Second, the problem of converting an NFA into a minimal equivalent NFA is NP-hard, even for NFAs over a unary alphabet [2, 3, 4].

In this paper, we explore the problem of searching for minimal NFAs. Our specific goal is to find all minimal NFAs recognizing a given unary language, where the language is given by specifying the first k bits of its characteristic function, which we represent as a bit vector. For example, the language $L = \{a^i \mid i \bmod 3 \neq 0\}$ could be represented by the bit vector `011011011011`, which indicates that $a^0 \notin L$, $a^1 \in L$, $a^2 \in L$, and so forth. Of course, this bit vector does not specify the language completely; hence we will actually search for minimal NFAs recognizing some *extension* of the given bit vector.

Note that our goal is quite different from that of works, such as [5, 6], that are aimed at *reducing the size* of an NFA without necessarily finding a *minimal* NFA.

We begin in Section 2 by recalling some preliminaries and exploring exhaustive search experimentally; we find that (with our hardware) exhaustive search is feasible only up to 6-state unary NFAs. Next, in Section 3, we develop a characterization theorem for NFAs, showing that a language is recognized by an n-state NFA iff it has what we call an *inductive basis* of size n. Then, in Section 4, we apply our characterization theorem as the foundation of a new incremental search algorithm for minimal unary NFAs. Rather than searching for NFAs directly, we instead search for inductive bases; this allows us to cut the search space dramatically. We find experimentally that our search can find 9-state unary NFAs in less time than exhaustive search requires for 6-state unary NFAs, even though the space of 9-state NFAs is tens of trillions of times larger than the space of 6-state NFAs.

2. PRELIMINARIES AND MOTIVATION

We recall some standard definitions, following [7]. Given a finite alphabet Σ, Σ^* denotes the set of all finite-length strings of elements of Σ, ϵ denotes the *empty string*, and xy denotes the *concatenation* of strings x and y. A *language* L is a subset of Σ^*.

Unlike [7], but like [8], we allow NFAs to have multiple start states and we do not allow ϵ-transitions:

DEFINITION 2.1. *An NFA M is a tuple $(Q, \Sigma, \delta, I, F)$ where*

- *Q is a finite set (the states),*
- *Σ is a finite set (the alphabet),*
- *$\delta : Q \times \Sigma \to 2^Q$ (the transition function),*
- *$I \subseteq Q$ (the set of initial states), and*
- *$F \subseteq Q$ (the set of final states).*

We often represent an NFA as a labeled directed graph with a vertex for each state and an edge labeled a from vertex q to vertex r whenever $r \in \delta(q, a)$; we indicate initial states with arrows and final states with double circles. For example, Figure 1 shows the graphical representation of a 5-state NFA.

The graphical representation allows us to define NFA acceptance quite simply; namely, NFA M accepts string w iff

ACM SE'06 March 10-12, 2006, Melbourne, Florida, USA

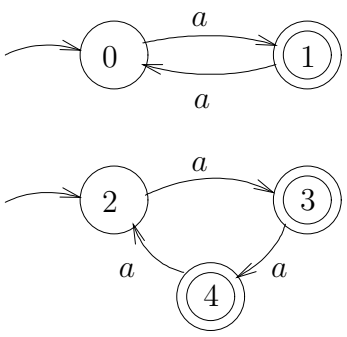

Figure 1: A 5-state NFA for $\{a^i \mid i \bmod 6 \neq 0\}$

there exists a path labeled with w from some initial state of M to some final state of M. This execution model is sometimes called *angelic nondeterminism*, because the NFA must "guess" the path that leads to acceptance. For example, the NFA of Figure 1 accepts *aaaa* because there is a path labeled *aaaa* from initial state 2 to final state 3:

$$2 \xrightarrow{a} 3 \xrightarrow{a} 4 \xrightarrow{a} 2 \xrightarrow{a} 3$$

The *language recognized by an NFA* is the set of all the strings that it accepts. For example, the language of this NFA is the set of all strings of a's whose length is not a multiple of 6, since $i \bmod 6 \neq 0$ iff $i \bmod 2 \neq 0$ or $i \bmod 3 \neq 0$.

What can we say about the minimum number of states of finite automata recognizing $\{a^i \mid i \bmod 6 \neq 0\}$? It is easy to show that the minimal DFA is a ring of 6 states. Also, we can see that the 5-state NFA of Figure 1 is not unique—we can construct a second 5-state NFA for the same language by interchanging the initial and final states, and reversing all the arrows. But can an NFA with fewer than 5 states recognize this language?

Along these lines, suppose we wish to determine the minimum number of states required by an NFA that recognizes the language $\{a^i \mid i \bmod 9 \neq 0\}$. The most straightforward approach is to try an exhaustive search. How many NFAs must be tried? Well, because this language is nonempty and does not contain ϵ, we know that I and F (the initial and final states) must be nonempty and disjoint. Hence the number of possible n-state NFAs is

$$\frac{n(n-1)}{2} \cdot 2^{(n^2)}.$$

(The first factor gives the number of choices for I and F, while the second gives the number of choices for δ.) Here is a table showing the number of n-state NFAs to be considered:

n	NFAs
1	0
2	16
3	1,536
4	393,216
5	335,544,320
6	1,030,792,151,040
7	11,821,949,021,847,552
8	516,508,834,063,867,445,248
9	87,042,659,012,253,300,578,844,672

A sequence of C programs was written to search exhaustively for NFAs recognizing $\{a^i \mid i \bmod 9 \neq 0\}$; the program `search`$n$`.c` searches for NFAs with n states. The programs were carefully optimized to use C bit operations as much as possible, and they were run on a 2.66 GHz Pentium 4. The following table shows the run times for `search`n`.c`, and the number of NFAs considered per second:

n	time	NFAs/sec
3	.000052 sec	30 million
4	.017 sec	23 million
5	16 sec	21 million
6	15 hours	19 million
7	20 years (?)	19 million (?)
8	860,000 years (?)	19 million (?)
9	145 billion years (?)	19 million (?)

The times up to $n = 6$ were measured experimentally, but of course the last three times are extrapolations. Notice that the number of NFAs considered per second decreases slowly as the number of states grows, so the extrapolated times are probably somewhat smaller than the actual times would be.

It turns out that the highest search completed, `search6.c`, found that no 6-state NFAs recognize $\{a^i \mid i \bmod 9 \neq 0\}$. And of course this language can be recognized by an 9-state DFA. But can it be recognized by a smaller NFA? Exhaustive search seems infeasible with present technology.

To do better, we need to develop smarter search techniques. To this end, we develop some useful theory in the next section.

3. INDUCTIVE BASES

In this section we establish a *characterization theorem* for NFAs. First we recall the notion of *prophecy* from [9]. The intuition is that when an NFA goes to a state q, it is "predicting" what the rest of the input will look like, assuming that the input is good.

DEFINITION 3.1. *Given NFA M and state q, the* prophecy of q, *denoted $P(q)$, is*

$$\{w \mid \textit{there is a path labeled } w \textit{ from } q \textit{ to some final state}\}.$$

Informally, the prophecy of q is the set of all strings that q "expects" to see.

For example, the states of the NFA in Figure 1 have the following prophecies:

$$\begin{aligned} P(0) &= \{a, a^3, a^5, a^7, a^9, \ldots\} \\ P(1) &= \{\epsilon, a^2, a^4, a^6, a^8, \ldots\} \\ P(2) &= \{a, a^2, a^4, a^5, a^7, \ldots\} \\ P(3) &= \{\epsilon, a, a^3, a^4, a^6, \ldots\} \\ P(4) &= \{\epsilon, a^2, a^3, a^5, a^6, \ldots\} \end{aligned}$$

It is easy to see that prophecies satisfy two key properties. First, the language of an NFA is the union of the prophecies of its initial states:

THEOREM 3.1. $L(M) = \bigcup_{q \in I} P(q)$.

Second, the different prophecies are related to each other. Recall first the definition of *left quotient*:

DEFINITION 3.2. $a\backslash L = \{w \mid aw \in L\}$

Now observe that $P(q)$ tells what q expects to see *now*, and $a\backslash P(q)$ tells what q expects to see *after* a. Hence we have the following relationship among the prophecies:

THEOREM 3.2. $a\backslash P(q) = \bigcup_{r \in \delta(q,a)} P(r)$.

Abstracting from these properties, we are led to introduce what we call an *inductive basis*. Let L be a language over alphabet Σ.

DEFINITION 3.3. *A collection $\mathcal{B}$ of languages is an* inductive basis for L *if*

- *There is a subcollection of $\mathcal{B}$ whose union is L.*
- *For each $B \in \mathcal{B}$ and $a \in \Sigma$, there is a subcollection of $\mathcal{B}$ whose union is $a\backslash B$.*

(The name "inductive basis" is chosen to reflect the fact that if $\mathcal{B}$ can generate a language C, then it can also generate $a\backslash C$.)

Using this concept, we can restate the two theorems above as follows:

THEOREM 3.3. *In any NFA M, the collection of prophecies of the states of M forms an inductive basis for $L(M)$.*

More interestingly, we can go in the opposite direction:

THEOREM 3.4. *If L has an n-element inductive basis, then L is recognized by an n-state NFA.*

Proof. Suppose that $\mathcal{B}$ is an n-element inductive basis for L. We can use $\mathcal{B}$ to construct an n-state NFA M recognizing L; we use the elements of $\mathcal{B}$ as the states of M. Define $M = (\mathcal{B}, \Sigma, \delta, I, F)$, where

- $\delta(B, a)$ is any subcollection of $\mathcal{B}$ whose union is $a\backslash B$.
- I is any subcollection of $\mathcal{B}$ whose union is L.
- F is the set of elements of $\mathcal{B}$ that contain ϵ.

The fact that $\mathcal{B}$ is an inductive basis for L exactly ensures that this construction can be done. (Notice however that the construction does not necessarily give a unique NFA.) To show that M recognizes L, the key property is that each state of M is its own prophecy:

LEMMA 3.5. *For all $B \in \mathcal{B}, P(B) = B$.*

Proof. We show that $w \in P(B)$ iff $w \in B$ by induction on $|w|$:

Basis:
We have $\epsilon \in P(B)$ iff $B \in F$ iff $\epsilon \in B$.

Induction:
Given $a \in \Sigma$, $u \in \Sigma^*$, we have $au \in P(B)$
iff $\exists B_f \in F$ such that there is a path from B to B_f labeled au
iff $\exists B' \in \delta(B, a), \exists B_f \in F$ such that there is a path from B' to B_f labeled u
iff $\exists B' \in \delta(B, a)$ such that $u \in P(B')$
iff $\exists B' \in \delta(B, a)$ such that $u \in B'$ (by the induction hypothesis)
iff $u \in a\backslash B$ (by the definition of δ)
iff $au \in B$. □

From the lemma, it follows that $L(M) = \bigcup_{B \in I} P(B) = \bigcup_{B \in I} B$. So, by the definition of I, M recognizes L. □

The construction of M in the above proof can be seen as a generalization of a construction given years ago by Conway: theorem 2 of chapter 5 of [10] builds the minimal DFA for a regular language L using the left quotients of L as states. This corresponds to an inductive basis $\mathcal{B}$ in which the subcollections used in making L and $a\backslash B$ all have size 1.

In summary, we have have the following characterization theorem:

THEOREM 3.6. *A language L can be recognized by an n-state NFA iff L has an n-element inductive basis.*

4. APPLICATION: FAST SEARCHES FOR MINIMAL UNARY NFAS

As an application, consider the special case where L is a *unary language* over alphabet $\{a\}$. If L is recognized by an n-state NFA M, then, by our characterization theorem, the collection of prophecies of the states of M forms an n-element inductive basis for L. Moreover, if M is minimal, then these prophecies are all distinct.

We can represent the situation concretely by using bit-vector representations for L and for the prophecies. This representation gives us efficient implementations of the set operations of interest to us. In particular, $a\backslash B$ becomes *left shift* and union becomes *bitwise "or"*.

For example, the language $\{a, a^2, a^4, a^5, a^7, a^8, \ldots\}$ becomes the bit vector `011011011`.... And $a\backslash\{a, a^2, a^4, a^5, a^7, a^8, \ldots\}$, which is $\{\epsilon, a^1, a^3, a^4, a^6, a^7, \ldots\}$, becomes `11011011`..., the left shift of the previous bit vector.

Another advantage of our framework is that we can get a *canonical ordering* of the states of M by putting the prophecies of M into increasing order lexicographically. For example, on the NFA of Figure 1, which recognizes

`011111011111011111...`

we get

```
010101010101010101...
011011011011011011...
101010101010101010...
101101101101101101...
110110110110110110...
```

These five rows correspond to states 0, 2, 1, 4, and 3, respectively. This canonical ordering is helpful in cutting the search space, since we no longer have to deal with permutations of the states of each NFA.

Given this framework, our search strategy is to search for a minimal NFA by searching for an inductive basis, represented as a set of rows as above. But, of course, we cannot search for infinite rows directly. So we introduce the concept of a *partial inductive basis*, consisting of the first i bits of each row, for some $i \geq 0$. This gives us n rows of length i with the following three properties:

Nondecreasing. They are nondecreasing lexicographically. (Note that they now need not be distinct.)

Makes L. The rows can generate the first i bits of L. That is, there is a subcollection of the rows whose bitwise "or" gives the first i bits of L.

Makes shifts. The first $i-1$ bits of these rows can generate the first $i-1$ bits of their left shifts. That is, for each row r, there is a subcollection of the rows such that the first $i-1$ bits of the bitwise "or" of the subcollection

gives the first $i-1$ bits of the left shift of r. (Notice that we can consider only the first $i-1$ bits of the left shift of r, since we know only the first i bits of r.)

Now suppose we are given the first k bits of L. Our approach is to search for a partial inductive basis of size s and width k incrementally. We maintain the invariant that `basis` holds a partial inductive basis of width `bw`, where $0 \leq$ `bw` $< k$. (Initially, `bw` $= 0$.) Our search tries to extend the width of `basis` in all possible ways, in depth-first order. To facilitate this, we let `newbasis` hold `basis` extended with one new column that we are trying, hoping to find a partial inductive basis of width `bw` $+ 1$. Specifically, we implement these data structures in C as follows:

- `basis` is held in the last `bw` bits of an `int` array of size s, where $0 \leq$ `bw` $< k$.
- `newbasis` is held in the last `bw` $+1$ bits of an `int` array of size s.

For example, when $L =$ `1001101111` we might have the following situation:

Notice here that `basis` is a partial inductive basis of size 4 and of width 4: it is nondecreasing, it can make the first 4 bits of L (in fact row 3 alone does the job), and it can make its left shifts (for instance, row 2 alone makes the first three bits of the left shift of row 1).

Given this setup, Figure 2 shows the main search loop of our program, `ibas.c`. In this code, `shrink(basis, newbasis)` uses right shifts to remove the last column from `basis` and `newbasis`, `increment(newbasis)` increments the last column of `newbasis`, and `grow(basis, newbasis)` uses left shifts to add a new column to `basis` and `newbasis`. (The new column of `newbasis` will contain 0's.)

When the loop exits, it means that there are no more partial inductive bases of size s and width k. This means that there are no more s-state NFAs that recognize even the first k bits of L. In this case, `ibas.c` checks whether any partial inductive bases of size s and width k have been found. If so, it exits; if not, it increments s and continues the search. In this way, it finds all of L's partial inductive bases of width k and minimum size.

On the other hand, it can be proved that each partial inductive basis of size s and width k that we find gives us an s-state NFA that recognizes some *extension* of the first k bits of L (which, of course, may not be the language that we had in mind!)

We can get a sense of the effectiveness of `ibas.c`'s search algorithm by running it when $L =$ `011111101111110`, corresponding to (the beginning of) the language $\{a^i \mid i \bmod 7 \neq 0\}$. In 12 seconds, `ibas.c` finds that there are no partial inductive bases of size less than 7, and finds 18 minimal partial inductive bases of size 7, corresponding to 18 distinct minimal 7-state NFAs, all of which recognize $\{a^i \mid i \bmod 7 \neq 0\}$. (All of the NFAs consist of a ring of 7 states, but they vary in their choices of initial and final states.)

Furthermore, profiling here shows that the main search loop iterates 209 million times, which means that 209 million candidate partial inductive bases are considered. But, as shown in Section 2, there are 11 quadrillion 7-state NFAs. So, in this case, `ibas`'s incremental search algorithm prunes the search space by a factor of 50 million.

Before further discussing the results obtained by `ibas.c`, we turn in the next subsection to a discussion of some details of our implementation.

4.1 Optimizations in the Implementation of `ibas.c`

We now describe some optimizations that we used to speed up the implementation of `ibas.c`. These optimizations were guiding by profiling the execution of `ibas.c` on a 2.66 GHz Pentium 4, using `gcc` and `gprof`. The specific times mentioned in the subsection are based on runs of `ibas.c` with $L =$ `011111101111110`. On this input, the original implementation of `ibas.c` took 25.3 seconds.

We begin with the implementation of `increment(newbasis)`. The original implementation kept an `int` counter and copied its bits, one by one, into the last column of `newbasis`, at a cost of 51 ns per call, or 39% of the total runtime. But of course most of the bits of the last column are usually unchanged! Implementing `increment` more carefully cuts the cost to 9 ns per call, reducing the total runtime by 32%. (Indeed, the problem of incrementing a binary counter is a classic example illustrating the *potential method* of amortized analysis [11, pp. 412–415].)

Next we consider the implementation of the check whether `newbasis` is a partial inductive basis of width `bw` $+ 1$. This requires three separate checks:

Nondecreasing. Is `newbasis` nondecreasing?

Makes L. Can `newbasis` generate the first `bw` $+ 1$ bits of L?

Makes shifts. Can `newbasis` generate its left shifts?

Note that the **Makes shifts** check amounts to checking whether `basis` can generate the last `bw` bits of `newbasis`. All of these checks can be implemented using fast C bit operations.

One simple optimization here is based on the observation that if `basis` is *strictly increasing*, then `newbasis` is guaranteed to be strictly increasing as well; in this case we can omit the **Nondecreasing** check. This optimization saves 5% of the original runtime. And using *sentinels* [11, p. 29] in the implementation of **Makes** L and **Makes shifts** saves 9% of the original runtime.

In total, we get a savings of $32 + 5 + 9 = 46\%$. And, indeed, the total runtime (when $L =$ `011111101111110`) is reduced from 25.3 to 13.8 seconds, a savings of 45.5%.

But we can make still one more optimization by considering the ordering of the three checks **Nondecreasing**, **Makes** L, and **Makes shifts**. The original implementation did the checks in this order. Of course this makes no difference as far as the correctness of the search is concerned, since `newbasis` must pass all three checks. But, interestingly, it can make a significant difference in efficiency, since

```
while (1) {
  if (the new column overflowed) {
    shrink(basis, newbasis);
    bw--;
    if (bw == -1)
      /* No more partial inductive bases of size s and width k. */
      break;
    increment(newbasis);
  }
  else if (newbasis isn't a partial inductive basis of width bw+1)
    increment(newbasis);
  else if (bw+1 == k) {
    /* Found a partial inductive basis of size s and width k. */
    print(newbasis);
    increment(newbasis);
  }
  else {
    grow(basis, newbasis);
    bw++;
  }
}
```

Figure 2: Main search loop of `ibas.c`

we can stop as soon as one of the checks fails. Profiling the (optimized) checks shows the following results:

- Each **Nondecreasing** check takes 5 ns and passes 88% of the partial bases.
- Each **Makes** L check takes 23 ns and passes 64% of the partial bases.
- Each **Makes shifts** check takes 46 ns and passes 1.3% of the partial bases.

This suggests that in fact **Makes shifts** should be done first, in spite of its cost, because it eliminates so many partial bases. Indeed, if we crudely assume that the results above hold when the tests are sequenced, then we can estimate the expected cost of the different possible orderings. For example, the cost per test of the original ordering **Nondecreasing**, **Makes** L, **Makes shifts** should be

$$5 + .88 \cdot (23 + .64 \cdot 46) \approx 51 \text{ ns},$$

while the cost per test of the ordering **Makes shifts**, **Makes** L, **Nondecreasing** should be

$$46 + .013 \cdot (23 + .64 \cdot 5) \approx 46 \text{ ns}.$$

(Experimentally, the latter ordering seems to be the fastest overall.) Since these checks account for about 70% of the optimized runtime, we would expect that switching from the former ordering to the latter should give an additional savings of 7%. In fact, the total runtime is reduced from 13.8 to 11.9 seconds, an additional savings of 13.8%.

4.2 More Results with `ibas`

In this subsection, we describe more of the results produced by `ibas`. The call `ibas 0111110111110` searches for NFAs recognizing (the beginning of) the language $\{a^i \mid i \bmod 6 \neq 0\}$. It produces the following output:

```
% ibas 0111110111110
No inductive bases of size 0.
No inductive bases of size 1.
No inductive bases of size 2.
No inductive bases of size 3.
No inductive bases of size 4.
Found an inductive basis of size 5:
  0010010010010
  0100100100100
  0101010101010
  1001001001001
  1010101010101
Found an inductive basis of size 5:
  0101010101010
  0110110110110
  1010101010101
  1011011011011
  1101101101101
There are 2 inductive bases of size 5.
```

The second of these inductive bases corresponds to the NFA in Figure 1, and the first to its reverse NFA; hence these two NFAs are minimal.

What happens if we specify less of L? It turns out that we get the same results all the way down to $L =$ `0111110`, but if we use $L =$ `011111`, the results change completely: we now get three inductive bases of size 2, corresponding to the NFAs shown in Figure 3. These NFAs all recognize aa^*, which is of course reasonable, given $L =$ `011111`. Note that the dotted arrows in the rightmost NFA are optional—they can be included or omitted without affecting the prophecies of the states.

Another question that we can consider is how the running time of `ibas` is affected by increasing the number k of bits of the language that are specified. Notice that doubling the value of k doubles the number of bits in the inductive basis, so we might imagine that this would roughly square `ibas`'s running time. But this is not so. For example, running

```
ibas 011111101111110
```

where $k = 15$, takes 12 seconds to find 18 minimal 7-element inductive bases. Running

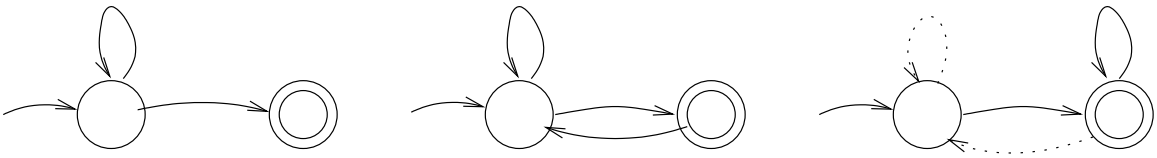

Figure 3: Three minimal NFAs for aa^*

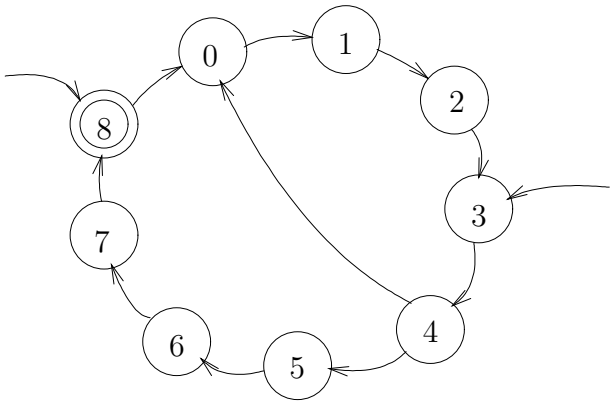

Figure 4: A minimal NFA for $(a^5 \cup a^9)^*$

```
ibas 01111110111111011111011111101
```

where $k = 30$, also takes 12 seconds to find the 18 minimal inductive bases. Interestingly, the number of candidate partial inductive bases increases from 209,473,372 to just 209,505,932, an increase of less than 0.02%. This follows from the incremental nature of the search—there are only 18 partial inductive bases of width 15, and it is only those that `ibas` tries to extend to width 30.

As a more ambitious example, we can consider the language $(a^5 \cup a^9)^*$. It turns out that the minimal DFA for this language has 33 states, because the longest string of a's that is *not* in the language is a^{31}. We can search for minimal NFAs for this language using

```
ibas 100001000110001100111001101111
```

In 68 minutes, this run finds that there are 27 minimal inductive bases, each containing 9 elements. The first of these bases corresponds to the 9-state NFA shown in Figure 4.

Finally, we can return to $\{a^i \mid i \bmod 9 \neq 0\}$, the language considered in Section 2. The call

```
ibas 011111110111111110
```

requires about 10 hours to find 56 minimal 9-element inductive bases. The time to search for inductive bases of each size is shown here:

n	time	growth factor
4	.001 sec	—
5	.035 sec	36
6	.86 sec	24
7	24 sec	28
8	14 min	34
9	10 hours	45

Recall from Section 2 that exhaustive search of 9-state NFAs for this language would require over 100 billion years.

5. CONCLUSION

Exploiting our inductive basis characterization theorem, we have dramatically improved upon exhaustive search for minimal unary NFAs. Interesting future directions include determining the number k of bits of the language that need to be specified in order to get the desired L, speeding up `ibas.c` by making use of additional structure in inductive bases, and trying to extend the approach to non-unary NFAs.

This work was partially supported by the National Science Foundation under grant HRD-0317692.

6. REFERENCES

[1] Hopcroft, J.E.: An $n \log n$ algorithm for minimizing the states in a finite automaton. In Kohavi, Z., ed.: The Theory of Machines and Computations. Academic Press, New York (1971) 189–196

[2] Stockmeyer, L., Meyer, A.: Word problems requiring exponential time. In: Proceedings 5th ACM Symposium on Theory of Computing. (1973) 1–9

[3] Jiang, T., Ravikumar, B.: Minimal NFA problems are hard. SIAM Journal on Computing **22** (1993) 1117–1141

[4] Gramlich, G.: Probabilistic and nondeterministic unary automata. In: Proceedings of Mathematical Foundations of Computer Science. Volume 2747 of Lecture Notes in Computer Science., Springer-Verlag (2003) 460–469

[5] Matz, O., Potthoff, A.: Computing small nondeterministic finita automata. In: Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (1995) 74–88

[6] Ilie, L., Yu, S.: Algorithms for computing small NFAs. In: MFCS '02: Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science. (2002) 328–340

[7] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)

[8] Perrin, D.: Finite automata. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics. Elsevier Science Publishers (1990) 1–57

[9] Arnold, A., Dicky, A., Nivat, M.: A note about minimal non-deterministic automata. Bulletin of the EATCS **47** (1992) 166–169

[10] Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall Ltd (1971)

[11] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. MIT Press (2001)